Saathi

Date __/__/____

Hashing Continue

Properties of Good Hashing :→ A hash function that maps every item into its own unique slot is known as a perfect hash function.

There is no systematic way to construct a perfect hash function. We can achieve a perfect hash function by increasing size of hash table.

Properties

(1) Efficiently computable.
(2) Should uniformly distribute the keys (Each table position is equally likely for each).
(3) Should minimize collisions.
(4) Should have a low load factor (number of items in the table divided by the size of the table.).

Complexity of calculating hash value using the hash function.

Time Complexity : O(n)
Space Complexity : O(1)

As we already know what is collision. When the hash function generates two same key for different data / big key at that moment collision occurs.

(1) Seperate Chaining (open Hashing)
(2) Open Addressing (closed Hashing)
   a → Linear probing
   b → Quadratic probing
   c → Double Hashing.

Page No.

Date __/__/____

Saathi

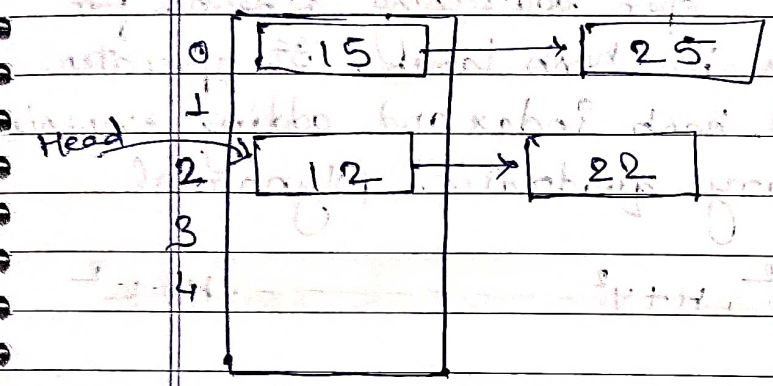
(1) Seperate chaining → The idea to make each cell of the hash table point to a linked list of records that have the same hash function. Chaining is simple but requires additional memory outside the table.

Ex → Hash function = key % 5, { Size → 0 → 4 }

Elements → 12, 15, 22, 25, 37.

12 % 5 = 2

22 % 5 = 2



(2) Open Addressing → All elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or not available space.

2.a) Linear Probing → The hash table is searched sequentially that starts from the original location of the Hash. If in case the location that we get is already occupied, then we check for the next location.

(1) Calculate the hash key key = data % Size.

(2) Check if hashTable[key] is empty or not.

if(empty) { hashTable[key] = data; }

Page No.

3. If the hash index already has some value then check for next index using

key = (key + 1) % size

4. Repeat until empty space is found.

(2.b) Quadratic Probing → Quadratic probing is an open addressing scheme for resolving hash collisions in hash tables. It operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial

$H+1^2, H+2^2, H+3^2, H+4^2 \text{ ---------- } H+K^2$

This method is also known as the mid-square method b/cause we look for i^2th probe (slot) in ith iteration and the value of $i = 0, 1, 2, \ldots, n-1$. We always start from the original hash location. If it is occupied then we check the other slot.

hash(x) % n is full then (hash(x) + 1^2) % n
(hash(x) + 1^2) % n is full then (hash(x) + 2^2) % n
This will repeat until ith iteration.

(hash(x) + $(i-1)^2$) % n is full. (hash(x) + i^2) % n.

(2.c) Double Hashing → It uses two hash function.

1. In first hash function $h_1(k)$ which takes the key & gives out a location on the hash table. If the

new location is not occupied or empty, then we can easily place our key.

(2) But in case the function is occupied we use secondary hash-function h2(k) in combination with h1(k).

h(k,i) = (h1(k) + i * h2(k)) % n.

↓

collision number (non-negative)

T.C → O(n)

* Load factor in Hashing?

The load factor of the hashtable can be defined as the number of items the hashtable contains divided by the size of the hash table. Load factor is the decisive parameter that is used when we want to rehash the previous hash function or want to add more element to the existing hash table.

It helps us in determining the efficiency of the hash function i.e. It tells whether the hash function which we are using is distributing the keys uniformly or not.

Load factor = Total element in HT / Size(HT)

{ HT → Hash Table }

Saathi

Date __/__/____

Rehashing → As the name suggest rehashing means hashing again. Basically when the load factor increase to more than its predefined value (the default value of the load factor is 0.75) the complexity increases so to overcome this the size of the array is increased (doubled) and all the values are hashed again and stored in the new double-sized array to maintain a low load factor and low complexity.

GfG

Application of Hash Data Structure

1.
2.
3.

Real Time Applications of Hash Data structure

1.
2.
3.
4.
5.

Advantages & DisAdvantages of Hash Data Struc

1.
2.
3.
4.
5.
6.

Date __/__/____ Brief of Sets, Multisets

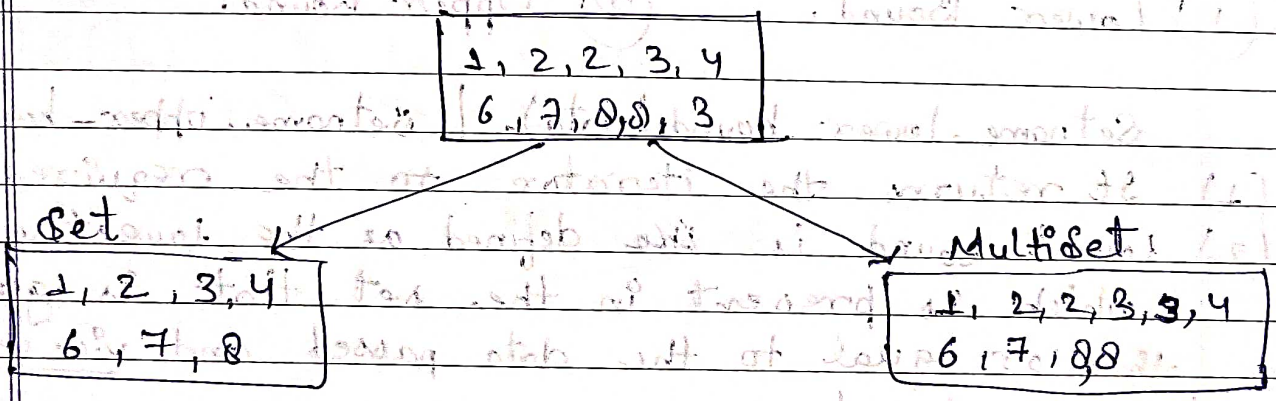
Saathi

As a mathematics, a set is a uniquely define collection of objects which cannot be repeated. Same functionality is in computer Science.

It has following property :-

1. A set is only able to store unique or distict data
2. If we store more than one occurrence of value in the set. It will still observe it as a single value
3. The data is stored in sorted order in the sets.

Types :→ Sets, MultiSets → It is able to store multi occurrence of the same data in sorted order.



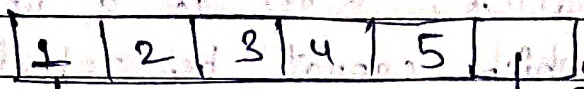
Declaration of Sets → Set<T> or multiset<T>

Pointers in Set :- Set do not have random access They does not follow the system of indexing.

1. Set has a start iterator called begin (Set.begin())
2. Set has a end iterator called end (Set.end()) which is equivalent to (n-1) in array where 'n' is size of array.

Date __/__/____

The end does not point to last element.



Operations → insert, Erase, (iterator) or Erase (iterator 1, iterator 2)
Size, etc.

* Bounds in Set.

Bound is defined as the lowest value greater than or equal to a particular number.

(1) Lower Bound. (2) Upper Bound.

Setname.lower_bound(data). | Setname.upper_bound(data)

(1) It returns the iterator to the required position.
(2) Lowest bound is defined as the lowest value which is present in the set that is greater than or equal to the data passed and vice versa for upper bound.
(3) If no such value is present, it returns an iterator to the end of the set.
(4) Can be used in both set & multiset.
T.C. O(Logn)

Ex → [1, 2, 4, 6, 7, 8].
set.lower_bound(3) ⇒ returns 4. because 3 is not present & lowest element which is greater than 3 is 4, present.

Page No.

Date __/__/____

Saathi

for (6) → 6 present. , (10) for 10 returns end iterator as greater than or equal to 10 is not present.

for upper bound →.

[1, 2, 4, 6, 7, 8].

1. (3) → 4 because 4 is the lowest, greater than 3.
2. (6) → is 7 because 7 is lowest element greater than 6.
3. (10) end iterator.
4. for (1) is (1) itself.
5. for (1) is 2.